VIA C3 Nehemiah Random Number Generator Application Note

Version 0.91

This is **Version 0.91** of the VIA C3 Nehemiah Processor RNG Application Note.

© 2003 VIA Technologies, Inc. All Rights Reserved.
© 2003 Centaur Technology, Inc. All Rights Reserved.

VIA Technologies and Centaur Technology reserve the right to make changes in its products without notice in order to improve design or performance characteristics.

This publication neither states nor implies any representations or warranties of any kind, including but not limited to any implied warranty of merchantability or fitness for a particular purpose. No license, express or implied, to any intellectual property rights is granted by this document.

VIA Technologies and Centaur Technology make no representations or warranties with respect to the accuracy or completeness of the contents of this publication or the information contained herein, and reserves the right to make changes at any time, without notice. VIA Technologies and Centaur Technology disclaim responsibility for any consequences resulting from the use of the information included herein.

VIA and VIA C3 are trademarks of VIA Technologies, Inc.

CentaurHauls is a trademark of Centaur Technology, Inc..

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Intel and Pentium are registered trademarks of Intel Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

LIFE SUPPORT POLICY

VIA processor products are not authorized for use as components in life support or other medical devices or systems (hereinafter called life support devices) unless a specific written agreement pertaining to such intended use is executed between the manufacturer and an officer of VIA.

1. Life support devices are devices which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. This policy covers any component of a life support device or system whose failure to perform can cause the failure of the life support device or system, or to affect its safety or effectiveness.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 INTRODUCTION 1-1
1.2 MAJOR FEATURES 1-2
1.3 IMPLEMENTATION VERSIONS 1-3
1.4 ABOUT THIS DOCUMENT 1-3
1.5 SUPPORT AVAILABLE 1-4

2 RANDOMNESS 101

2.1 THE NEED FOR RANDOM NUMBERS 2-1
2.2 WHAT IS A RANDOM NUMBER 2-3
2.3 HOW TO GENERATE RANDOM NUMBERS 2-5
2.3.1 SOFTWARE GENERATORS 2-5
2.3.2 PHYSICAL SOURCES 2-8
2.3.3 TRUE HARDWARE GENERATORS 2-9
2.4 HOW TO VERIFY RANDOMNESS 2-10
2.5 THE INTEL 82802 RNG 2-11

3 PROGRAMMING INTERFACE

3.1 GENERAL ARCHITECTURE 3-1
3.2 IDENTIFICATION & ENABLING THE RNG 3-3
3.2.1 OVERVIEW OF ENABLING 3-3
3.2.2 CPUID INSTRUCTION IDENTIFICATION 3-3
3.2.3 MSR 0X110B 3-4
3.3 NEW X86 XSTORE INSTRUCTION 3-5
3.3.1 XSTORE: STORE RANDOM DATA INSTRUCTION 3-5
3.3.2 XSTORE: NUMBER OF BYTES STORED 3-7
3.3.3 REP XSTORE INSTRUCTION 3-7
3.4 STRING FILTER FUNCTION 3-8
3.4.1 BASIC BEHAVIOR 3-8
3.4.2 MULTITASKING ISSUES 3-9
3.5 STARTUP STATE & ACTIONS 3-10
3.5.1 RESET DEFAULTS 3-10
3.5.2 INIT ACTIONS 3-10
3.5.3 RNG ENABLE 3-10
3.5.4 RNG DISABLE 3-10
3.5.5 WRITING THE MSR 3-11
3.6 MANUFACTURING TESTS 3-11

4 PROGRAMMING CONSIDERATIONS

4.1 RANDOMNESS CONTROL 4-1
4.2 STARTUP TESTING 4-2
4.3 TIME-OUT ISSUES 4-2
4.4 DC BIAS 4-3

5 SUMMARY HARDWARE DESCRIPTION

5.1 RANDOM BIT GENERATOR 5-1
5.2 DATAPATH & MICROCODE FUNCTIONS 5-2

6 PERFORMANCE & POWER

6.1 RANDOM BIT GENERATION SPEED 6-1
6.2 NEW INSTRUCTION PERFORMANCE 6-2
6.2.1 XSTORE INSTRUCTION 6-2
6.2.2 REP XSTORE INSTRUCTION 6-2
6.3 POWER MANAGEMENT 6-3

7RANDOMNESS TESTING

7.1 INTRODUCTION TO TESTING RANDOMNESS 7-1
7.1.1 GENERAL BACKGROUND 7-1
7.1.2 SOME IMPORTANT TESTS 7-4
7.1.3 NIST TEST SUITE 7-5
7.1.4 FIPS 140 7-6
7.1.5 DIEHARD TESTS 7-7
7.2 OUR TESTING METHODOLOGY 7-7
7.3 SOME SUMMARY RESULTS 7-8
7.3.1 FIPS-140 TESTS 7-8
7.3.2 DIEHARD TESTS 7-8
7.3.3 NIST TESTS 7-8
7.4 TESTING CONCLUSIONS 7-8

CHAPTER

1

INTRODUCTION

This chapter introduces the new VIA C3 Nehemiah processor Random Number Generator (RNG) and summarizes its major features. In addition, a summary description of the document contents and reader assumptions is included.

1.1 INTRODUCTION

The recent explosive growth of PC networking and Internet-based commerce has significantly increased the need for good computer and network security mechanisms. Good security requires good random numbers. Which can be a problem—it is actually quite hard to generate *good* random numbers on a computer. (For those who don't understand what a random number is, why they are needed in computer applications, what is a "good" one is, and how they are generated, Chapter 2 includes some background on these topics.)

To address this need for good random numbers in security applications, the new VIA C3 Nehemiah processor includes a high-performance hardware-based random number generator (RNG) *on the processor die*. This RNG uses random electrical noise on the processor chip to generate highly random values at an extremely fast rate. It provides these numbers directly to the application via a new x86 instruction that has built-in multitasking support. By including this unique capability directly within the processor, the VIA C3 Nehemiah processor helps enables a new level of security for personal computers and embedded devices.

1.2 MAJOR FEATURES

The RNG feature has many powerful and unique features.[1] In particular:

High Quality

- **Hardware-Based.** Truly random numbers require a non-deterministic source (as opposed to a programmed algorithm). The VIA C3 Nehemiah random bits are obtained from electrical random noise on the processor chip and meet the requirements of truly random numbers (statistically good, unpredictable, and unreproducible).
- **Multiple Quality vs. Bit Rate vs. Processor Load Options.** Application code can control a hardware filter that selects and returns one-of-n bits generated (to reduce potential bit-to-bit correlation). Four levels of quality filtering can be selected. The slowest setting is required for best randomness of the returned values. The faster settings, however, provide very good entropy for applications that use the generated numbers as a seed to another algorithm that which produces the random value to be used.
- **Designed for Verification and Test.** Because the RNG allows direct access to the entropy source, third parties can perform their own tests of the output. In addition, Centaur Technology has performed extensive testing of the RNG feature using a variety of tests and analysis. This data is available to software developers. Comparison data is also provided on some alternate sources of randomness.

High Performance

- **High Bit Rate.** The RNG produces bytes at a variable rate. Typical rates vary from approximately 800K bits per second at the best quality rate to 6M bits at the fastest "good entropy" setting. This is from 10 times to 80 times faster than the next fastest RNG hardware solution that we know of—the Intel RNG included in the 82802 Firmware Hub. Higher rates address the needs of applications requiring high bit rates and can be used with software that algorithmically increases the quality (randomness) of the numbers produced, for example by applying further hashing algorithms to the output.
- **Asynchronous Multi-byte Generation.** The hardware generates random bytes at its own pace. These accumulate automatically into four 8-byte hardware buffers with no impact to program execution cycles. Software can read the accumulated bytes at any time. This asynchronous approach allows hardware to generate large amounts of random numbers completely overlapped with program execution. This is opposed to good software generators, which can be fast but consume a significant number of cycles.

An Efficient and Secure Programming Interface

- **Application-level Access:** Application code can store the random numbers directly from the hardware, without the intervention of privileged device drivers. A new x86 instruction—`xstore`–is provided to store random data and the associated valid byte count. In addition, a `REP xstore` instruction allows large amounts of random data to be collected with only one instruction. Direct access ensures that applications receive data directly from the random number generator, avoiding the possibility of receiving nonrandom data due to malicious or accidental device driver flaws.

[1] Patents pending.

- **Transparent to the Operating System:** The operations involved in obtaining random bytes and verifying the configuration operate in user mode, without operating system knowledge or involvement. For security purposes, changes to the configuration require a higher privilege level.
- **Inherent multitasking (Atomic Instruction):** Storing random data is a single atomic instruction within an application: that is, a `xstore` returns both the data and the size of the random data, as well as clearing the hardware data register. If a task-switch occurs, and another application wants to store random data, it gets completely new random data. The `REP xstore` instruction is interruptible similar to x86 string instructions in atomic increments.

Many Optional Features

- **Contiguous String Filter:** This is an optional function that has may have applicability in some environments. It is also useful for testing the RNG. When enabled it ensures that no contiguous string of zeroes or ones longer than a certain user-specified number (up to 64) can be generated.
- **Raw Bit Access.** For testing and characterization purposes, the raw internally generated random bits may be accessed. Normally the raw bits are filtered through a "whitener" to obtain desirable random characteristics.
- **Controllable Voltage Bias.** For testing and experimentation purposes, the internal voltage bias to the various oscillators can be adjusted.

1.3 IMPLEMENTATION VERSIONS

This datasheet defines the RNG function as implemented in the "stepping 3" and later versions of the VIA C3 Nehemiah processor. One earlier stepping of the VIA C3 Nehemiah processor was shipped in very limited quantities that does not contain a properly functioning RNG feature. Software should not attempt to use the RNG feature on these earlier processor versions.

The detailed CPUID identification of the first VIA processor containing the RNG feature is:

Vendor ID: CentaurHauls
Type: 0
Family: 6
Model: 9
Stepping: 3 (also called EC-0H)

It is theoretically possible that even on the correct stepping, the RNG feature may not be present (see section 3.6 for reasons). Thus, after the correct processor version is identified, one must still check for existence of the RNG feature as described in section 3.2.

1.4 ABOUT THIS DOCUMENT

This document has sections aimed at two different levels of readers:

- Those who do not know much about random numbers, their generation, their testing, and their use in security applications. For these readers, we have included some summary

background information on relevant topics. More experienced readers may wish to skip these sections, but the detailed technical sections assume this general background knowledge.

- Those who know random number basics and actually want to use (program) the VIA C3 Nehemiah RNG feature. For these readers, detailed information on registers, instructions, etc. are provided. The detailed sections assume that these readers are fluent in x86 assembly-language specifics as well as the random number concepts discussed in the background sections.

This application note comprises the following chapters:

- **Chapter 1, Introduction.** This introductory chapter.
- **Chapter 2, Random Numbers 101. A** background tutorial on the need for, the definition of, the methods to generate random numbers, and a little about testing for randomness. This chapter is optional, but this level of information is generally assumed in the subsequent technical descriptions.
- **Chapter 3, Programming Interface. A** detailed description of the program-visible registers and instructions associated with the RNG feature. This is the "specifications", per se.
- **Chapter 4, Programming Considerations.** Some discussion and hints about programming considerations for the RNG feature.
- **Chapter 5, Summary Hardware Description.** A summary description of the hardware mechanisms of the RNG. This information is not required in order to use the RNG, but may be useful for those interested in a detailed analysis of the output.
- **Chapter 6, Performance and Power.** Description of RNG performance as well as power management considerations.
- **Chapter 7, Testing for Randomness.** A background tutorial on randomness testing theory followed by a description of our testing methodology and a summary of our results. [Results are not provided in this version of the document.]

1.5 SUPPORT AVAILABLE

For operating systems developers and applications developers who might need assistance in using the VIA C3 Nehemiah RNG feature, we provide direct consultation with the RNG developers and testers. In addition, some source code can be provided:

- An assembly-language code snippet to perform either a FIPS 140-1 or FIPS 140-2 test. This code is self-contained and can be easily configured for either test's (or your own) limits.
- Linux device source code that uses the VIA C3 Nehemiah RNG feature to provide random numbers analogous to the /dev/random driver.

For information about direct access to the VIA C3 Nehemiah RNG designers, software developers should contact:

rng_support@centtech.com

CHAPTER

2

RANDOM NUMBERS 101

This chapter provides optional background material on random numbers for computers. Topics covered are the requirements for random numbers, characteristics of good random number generators, the range of solutions, how to test for randomness, etc.

For those who do not already know what random numbers are, what applications use them, how they are generated, and why most current generators do not really produce random numbers, this chapter provides some general background on these topics. This material is optional reading; some of the subsequent detailed descriptions of the VIA C3 Nehemiah RNG, however, assume at least this much knowledge of random numbers.

This chapter also includes a summary comparison with the only similar hardware RNG: the Intel RNG contained in the 82802 Firmware Hub.

[Note to real mathematicians, statisticians, security experts, and information scientists. This background introduction intentionally simplifies some very complex technical topics and skips many important considerations.]

2.1 THE NEED FOR RANDOM NUMBERS

To help understand the definition of a random number for computer applications, we need to understand something about how they are used. Historically, many computer applications have required random numbers as an essential component of their logic. For example,

- Monte Carlo simulations of physical phenomena in physics, chemistry, mechanical engineering, electrical engineering, meteorology, statistics, aerodynamics, hydrodynamics, and other similar disciplines.
- Numeric analysis solutions for many mathematical problems such as solving integral and differential equations, solving linear and non-linear equations, etc.
- Casino games and online gambling (to simulate card shuffling, dice rolling, etc.)
- Creation of lottery numbers
- Generating data for statistical analysis
- Generating data for psychological testing
- Computer games (choosing random behaviors, dealing cards, etc.)
- Computer music generation

These applications have different requirements for random numbers. For example, computer games generally require neither high performance in generating random numbers nor particularly high degrees of randomness. Other applications, such as psychological testing require higher degrees of randomness, but do not require high performance generation. Large-scale Monte Carlo-based simulations, however, use millions, billions, or more random numbers, and thus have very high performance requirements.[2] These applications also require good statistical properties of the random numbers, but unpredictability is not particularly important. Other applications, such as online gambling, have very stringent randomness requirements as well as stringent non-predictability requirements (the next number cannot be easily predicted from the previous numbers).

While these historical applications are important to PCs, today the major need for "high-quality" random numbers is *computer security*. The recent explosive growth of PC networking and Internet-based commerce has significantly increased the need for a wide variety of computer security mechanisms. High-quality random numbers are essential to all of the following major components of computer security:[3]

- **Confidentiality.** Data encryption is the primary mechanism for providing confidentiality. Many different encryption algorithms exist (symmetric, public-key, one-time pad, etc.), but all share the critical characteristic that the *encryption/decryption key must not be easily predictable*. The cryptographic strength of an encryption system depends on the strength of the key: that is, the difficulty of predicting, guessing, or calculating the decryption key. To ensure that keys cannot be easily guessed, random number generators are used to produce cryptographic keys and other secret parameters in virtually all serious security applications. Many public key encryption protocols, such as RSA encryption using PKCS #1, also require reliable sources of randomness.

[2] James E. Gentle, *Random Number Generation and Monte Carlo Methods,* Springer, 2002

[3] This document assumes some knowledge of security concepts, functions, and protocols. If you need some technical background on these areas and more about random numbers, we recommend the following three "classics":

Bruce Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, John Wiley & Sons, 1996.

Alfred J. Menezes et al., Handbook of Applied Cryptography, CRC Press, 1997

Donald E. Knuth, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms,* AddisonWesley, 1998. Chapter 3 covers random number theory in some detail.

Many successful attacks against cryptographic algorithms have focused not on the encryption algorithm but instead on its source of random numbers. As a well-known example, an early version of Netscape's Secure Sockets Layer (SSL) collected data from the system clock and process ID table to create a seed for a software pseudo-random number generator. The resulting "random" number was used to create a symmetric key for encrypting session data. Two graduate students broke this mechanism by developing a procedure for accurately guessing the random number in less than a minute and thus guessing the session key.[4]

- **Authentication.** Challenge/response authentication protocols require that "challenge" values be unpredictable to ensure that attackers cannot reply data from previous authentication transactions. The strength of passwords used to protect access and information also depends on the difficulty of predicting or guessing the password. As a result, strong random number generators are necessary to automatically generate strong passwords.
- **Integrity.** Digital signatures and message digests are used to guarantee the integrity of communications over a network. Random numbers are often used in digital signature algorithms to make it difficult for a malicious party to forge the signature. Many signing algorithms, including the U.S. Government's Digital Signature Standard[5] also require random sources to ensure the security of the signing keys.

In summary, good security requires good random numbers.

2.2 WHAT IS A RANDOM NUMBER

The notion of a (good) random number is a non-intuitive concept. What is randomness? Can any *single* number be considered "random"? Can any *finite sequence* of numbers be considered "random"? Is it the *numbers* themselves, or the *process* that generated the numbers that defines randomness? These questions are the subject of many technical papers, doctoral theses, and bar-fights among philosophers, mathematicians, statisticians, and information scientists.

One essential fact emerges from this work: "…there is no such thing as a random number—there are only methods to produce random numbers".[6] A number by itself (or any finite sequence of numbers) is not random, or non-random. (Is zero any less random than 42?) A practical definition of randomness must focus on the characteristics of the random number generator as well as the characteristics of numbers it generates.

To identify the desired characteristics, we consider the theoretical definition of a perfect random number generator. A *truly random number* generator is one where the probability of each generated bit being a zero (or one) is exactly ½, regardless of what it has generated before.[7] For example, an unbiased coin flip should generate truly random numbers. This definition has many implications beyond the statistical characteristics of the bits generated. It implies that the generator is *unpredictable* (one can't predict the next coin toss better than ½ the time, and *unreproducible* (two separate coin flipping experiments give independent results).

[4] http://www.demailly.com/~dl/netscapesec/wsj.txt, or
http://www.ddj.com/documents/s=965/ddj9601h/9601h.htm

[5] "Digital Signature Standard (DSS)", Federal Information Processing Standard Publication 186, May 19, 1994.

[6] John von Neumann

[7] There are many different definitions used in the literature, but they are all equivalent to this definition.

Entropy is a related term that we will use a lot in this document. In general, entropy is the measurement of randomness or chaos. Something that has high entropy is chaotic and unpredictable. Something that is ordered has low entropy. In cryptography, entropy is usually measured in bits. A truly random one-bit number has an entropy of one bit. Entropy is directly connected to the information content of a message. For example, a sequence of *n* random bits (i.e., a sequence having an entropy of *n* bits) cannot on average be expressed or defined in less than *n* bits.

The problem with the theoretical definition of randomness is that the amount of "entropy" cannot be measured perfectly. It can be relatively easy to show that a particular generator is *not* random based on its design (for example, its design causes the same sequence of generated numbers to reoccur ever so often). Unfortunately, demonstrating randomness is much more difficult. Indeed, it is impossible to *prove* that the output from a given random number generator is truly random. As a result, the best we can do is to analyze the statistical characteristics of the previous generated bits and compare them to a mathematically perfect generator.

For security applications. the most important characteristics of random numbers are (paraphrasing Schneier[8]):

- **Unbiased Statistical Distribution.** Truly random numbers have certain statistical characteristics. As discussed in chapter 7, these characteristics can be calculated for a theoretical random generator and compared to the statistics for a generated sample. Based on this comparison, the probability that the sample came from a biased source, can be calculated.

 For example, one important statistical characteristic is the number of 0 or 1 bits in the sample. For a sample of 20,000 bits from a truly random generator, the probability is 0.0001 that the number of 1 bits is less than 9,725 or more than 10,725.[9] Thus, if we analyze 10,000 samples of 20,000 bits each, and find many samples that deviate from these limits, we should suspect that our generator is not truly random (we should expect only one deviation out of 10,000 tests from a truly random generator).

 Of course, this test does not prove randomness. For example, if our generator is stuck at generating alternate 1s and 0s, it would pass this monobit test even though the output is clearly non-random. Thus, many different statistical characteristics must be analyzed in order to feel comfortable about a generator. Our alternating 1s and 0s sequence would quickly fail many other tests such as a bit-to-bit correlation test, or a multi-bit block frequency test. More information about specific tests is provided in Chapter 7.

 Security applications minimally require "good" statistical characteristics of the generated random numbers. A generator that meets only this non-biased requirement is called *pseudo-random*.

- **Unpredictability**. In addition to good statistical properties, truly random numbers must be unpredictable; that is, the probability of correctly guessing the next bit of a sequence of bits should be exactly ½ regardless of the values of the previous bits. Some applications do not care about this unpredictability characteristic, but it is critical to security applications.

 An example of a predictable sequence that has good statistical characteristics would be a generator that produces, for example, 20,000 statistically good random bits and then re-

[8] Schneier (1996) p.45-46

[9] This specific example is one we expand upon in chapter 7 as an example of statistical testing of random numbers.

peats the exact sequence again. To identify this problem would require analyzing the samples larger than 20,000 bits. A worse problem with predictability is the use of typical software generators. For a software generator to be unpredictable requires that the software algorithm or its initial values be "hidden". (If we know the algorithm and initial values, we can produce the same sequence as the software generator.) From a security viewpoint, a "hidden algorithm" or "magic number" approach is very weak if it is possible for someone to predict the output. For example, the previous section mentioned a well-known case where the security of a predictable (software) random number generator failed because the starting values could be guessed easily.

A generator which deterministically derives its output from a starting value, but which meets both the statistical and unpredictability requirement is called *cryptographically secure pseudo-random*. (To be precise, these RNGs meet the unpredictability requirement by making it *computationally infeasible* to predict output bits. An adversary with unlimited computational power could guess possible values for the starting state until one is found that matches the output.)

While some statistical tests may indicate a degree of predictability, whether a generator is considered unpredictable is primarily based on it's inherent design.

- **Unreproducibility.** To meet this requirement, two of the same generators (or the same generator at two different times), given the same starting conditions, must produce different outputs. Clearly, software algorithms do not meet this requirement; only a hardware generator based on random physical processes can generate values that meet this stringent security requirement. There no test to determine unreproducibility—this determination is based on the inherent design of the generator.

A generator that meets all three requirements is called *real,* or *truly, random*.

2.3 HOW TO GENERATE RANDOM NUMBERS

We now know what random numbers are and why they are critical to many applications, especially security. Why do we need anything special, however, to generate them—isn't there a random number generator included in every programming library? Well, there is, but as Schneier points out: "…these [software] random number generators are almost definitely not secure enough for cryptography, and probably not even very random. Most of them are embarrassingly bad."[10] This section explores this problem and describes the three basic approaches to generating random numbers—software, physical sources, and quantum randomness—and some combinations of these techniques.

2.3.1 SOFTWARE GENERATORS

The most common approach to generating random numbers is using a deterministic algorithm implemented by a computer program. Such deterministic algorithms cannot generate truly random numbers. (At best they are predictable and reproducible, and at worst, have bad statistical characteristics.) The quality of software-based random number generators is often summarized with John Von Neumann's famous quote: "Anyone who considers arithmetic methods of pro-

[10] Schneier (1996) p44

ducing random digits is, of course, in a state of sin."[11] Thus, software generators are usually called *pseudo-random* or *quasi-random* generators. There is, however, a wide range of "quality" among software generators with some being considerably better for security purposes than others.

Many different software algorithms are used, but all comprise two basic steps for each value generated:

1. **Choose a starting value or "seed".** All software generators start with some initial value, or *seed*. Some set of arithmetic operations (the mixing algorithm) are performed on this initial seed to produce the first random result. In most cases, this result is then used as the seed to produce the second result, and so forth. In some cases, a new seed is constructed for each value based on external (to the random number generator) events (for example, the time-of-day clock).
2. **"Mix up" the bits from the seed in an attempt to increase its *apparent* entropy.** The mixing algorithms can be grouped into two distinct groups. Conventional generators such as typically provided for the C language `rand()` function perform simple arithmetic operations. Newer algorithms designed specifically for cryptography perform much more complicated (and time-consuming) operations. Each algorithm has different performance characteristics (bit-generation rate), and different randomness characteristics. Although the mixing operation does not technically increase the total entropy (deterministic process cannot create entropy), it is believed that properly-designed mixing operations can make it *computationally infeasible* to distinguish the output from a true random source.

Let's look at the conventional approach, which represents most extent generators. These algorithms take the previous value generated and they apply some arithmetic operation to it to produce the next value. Historically, little theory was applied to the mixing algorithm—if it seemed to mix up the seed bits a lot, it was considered a good algorithm. As an example, Knuth references an *extremely* complicated algorithm he invented in 1959 that, when executed, quickly converged to a constant value. This led him to conclude: "random numbers should not [be] generated with a method chosen at random. Some theory should be used."[12] Subsequently, a significant amount of theory has been applied to defining and analyzing good mixing algorithms. The result is a set of *linear* mixing algorithms that are relatively fast and have reasonably statistical characteristics.

A basic example of a linear algorithm is widely used in modern software generators. It is the *linear congruential method*. The calculation is simple:

$$x_{n+1} = (ax_n + c) \bmod m$$

where the constants a, c, and m meet some very specific mathematical constraints and the calculations are performed in the word size of the processor. If the constants are chosen correctly, then the cycle length of the generator is m. That is, after m numbers, the generated sequence repeats itself. This is, for example, the approach used in the Microsoft Visual C++ 5.0 library implementation of `rand()` where $m = 2^{31}$ (0x80000000).

[11] John Von Neumann (1951). Quoted in Knuth, 1968, Vol. 2

[12] Knuth, pg 6

There are several other linear algorithms that have good theoretical underpinnings. These theoretical characteristics are not intuitively obvious, however. Inventing your own algorithm or making minor changes to a known good algorithm can yield very bad randomness.

Although the cycle length of the Microsoft `rand()` example is too short for Monte Carlo simulations and other applications requiring large amounts of random, linear algorithms *can* have a reasonably long cycle. Similarly, well-designed linear algorithms *can* generate numbers with reasonable statistical characteristics. In general, however, the most common software algorithms do not have statistical characteristics as good as truly random values.

Even worse, these algorithms are easily predictable; that is, knowing the algorithm and the constant values (which are often fixed in the software), one can predict the entire sequence. Even if the seed value isn't constant, it can be guessed or, knowing the last generated value (or any other previously-generated value), one can trivially predict the sequence. These generators are also reversible: given the algorithm and a value, previous values can be easily calculated. These cyclical and predictable characteristics are a disaster for security applications.

The good news is that extensive research, publication, and analysis has been done for years on software random number algorithms for use in security applications. One result is the design of some improved non-linear software generators that closely match the statistical characteristics of a truly random generator and are somewhat unpredictable. These are called *cryptographically secure pseudorandom bit* generators (CSPRBG). They use large internal calculation sizes (typically 64- to 1024-bits) and use cryptographically strong "one-way" algorithms. They use values that are determined dynamically rather than built into the code.

Strong one-way functions make it is impracticable to calculate subsequent or previous values given the current value unless the generator's state is known. Even if an adversary discovers the state at a particular point, the algorithms are not reversible, making it infeasible to determine the number prior to the compromise. Of course, knowing the state of the RNG does allow future results to be calculated. As a result, these algorithms rely on secrecy of the initial conditions and computational intermediates to ensure unpredictability.

A simple example of such a CSPRBG is the Blum-Blum-Shub generator.[13] Here is an abstract:

1. Start by selecting two large primes, p and q, with certain characteristics and calculate the hidden value $n = pq$. Choose some seed, x_0, with certain characteristics relative to n.

2. Calculate each new random value by
 $x_{i+1} = x_i^2 \bmod n$
 $z_{i+1} = \textit{the least significant bit of } x_{i+1}$

Even if n is known, its particular numeric characteristics make the difficulty of calculating the previous value very high.

As good as these CSPRBGs are compared to conventional linear software algorithms, however, they still have some problems:

- They are not widely available. They are also prone to typical software mistakes in the implementation.
- They are often computationally very slow.

[13] (In choosing an example, I couldn't resist this name.) See Menezes et. al, pg. 186

- They are predictable in the sense described above (i.e., forward prediction if the initial conditions are known).
- Like all software algorithms, they are reproducible—they produce the same results starting with the same input
- Their security relies on an assumption that adversaries do not have enough computational power to perform certain mathematical operations. As a result, their irreversibility cannot be proved, although, in practice, this is not a major problem.

There is a way to improve the situation here. Rather than relying solely on a software generator to produce all of the randomness, a software generator can be used in conjunction with a source of true randomness (entropy). Instead of using the previous result as the seed for the next number to be calculated, output from the independent source of entropy can be used as the seed. Even if the entropy source is imperfect (provides less than one bit of entropy per output bit), the results in this case can be more robust than what could be generated by either the software or the entropy source alone.

An example is using a hardware random number generator (see the next section) to produce the seed for a software algorithm. Although a variety of designs are used, the software algorithms are often based on cryptographic *hash functions*. A hash algorithm mixes up its input bits in a fashion such that its output bits should be unpredictable unless the entire input is known. Thus, a good hash algorithm effectively destroys any statistical correlations among the input bits and makes it computationally infeasible to recover the input from the output. The most commonly used cryptographic hash algorithm, the *Secure Hash Algorithm (SHA-1)*[14] is a U.S. government standard and has been extensively analyzed.

The primary reasons for using a hybrid approach (as opposed to a pure hardware-based or pure entropy-based approach) are (1) the historically slow speed of entropy and hardware generators, and (2) the lack of perfect randomness in the hardware or entropy generators. While the VIA C3 Nehemiah processor RNG provides both high performance and high quality output, users may choose to use it in conjunction with software-based algorithms. (In its full quality mode, it is very fast and has such good randomness that combining it with software is generally not necessary. If one prefers hash-based generation, for example to obtain higher net bit rates, the VIA C3 Nehemiah RNG has a faster "entropy" mode that produces data that is well suited as input to hash function based generators.

2.3.2 PHYSICAL SOURCES

In terms of randomness characteristics, *entropy generators* based on physical phenomena fall in between software generators and quantum-based hardware generators. The idea here is to sample some "random" physical process and assume the samples are truly random. Unfortunately, computers and software tend to be very predictable, so designers must be very careful to ensure that they collect good entropy.

For example, a widely used PC encryption program derives its keys from several seconds of mouse movements and keystroke timing (directed by the program). The Linux operating system has random number generators (/dev/random and /dev/urandom) that use entropy generated by the keyboard, mouse, interrupts, and disk drive behavior as the seed. Microsoft's CryptGenRandom function (part of the Microsoft CryptoAPI) is similar. It uses "…—for in-

[14] FIPS 180-1, Secure Hash Standards, U.S. Department of Commerce/NIST, 1995

stance, mouse or keyboard timing input—that are then added to both the stored seed and various system data and user data such as the process ID and thread ID, the system clock, the system time, the system counter, memory status, free disk clusters, the hashed user environment block" as the seed Many other similar environmental sources of randomness have been tried.

While these physical activities may look "random", their randomness cannot be proven, and they run the risk of generating poor entropy (or no entropy) if the sampled physical activity is dormant or repetitive. There are several potential security "holes" with using such physical activities. For example, in networked applications such as browsers, the application traffic between a client and server effectively "publishes" the locations and sequence of the client's mouse-events. Similarly, users may enable "snap-to" options that center the mouse pointer in the center of the button to be pressed and make the click locations predictable. As a result, the entropy from mouse movements in these environments could be far less than an RNG designer expected. Similarly, asking the user to create entropy using the keyboard creates bias since humans tend to follow certain patterns in typing (such as a tendency to use the center of the keyboard).

In summary, it is dangerous to use the entropy values *directly* as random numbers. Instead, the entropy values are usually used as the seed to a good hash algorithm to produce the final random numbers. (In the Microsoft CryptoAPI and Linux examples, a SHA-1 algorithm is applied to the entropy seeds). Other common problems with computer entropy generators on computers are that they require hooks in the operating system, they are difficult to test, they often require some user involvement, and they are slow (since they are based on macro physical events).

An unusual example of a physically-based random source is available through www.random.org, which continually provides generated random numbers using atmospheric noise (sampled as radio frequency noise). The rate varies, but this generator typically produces about 40,000 bits per second.

2.3.3 TRUE HARDWARE GENERATORS

By now you should be convinced that good security requires good random numbers, and while it is possible to generate pretty good statistically random numbers using software, it is difficult, slow, and subject to numerous security pitfalls. The only truly random generator is some mechanism that detects quantum behavior at the sub-atomic level. This is because randomness is inherent in the behavior of sub-atomic particles (remember quantum mechanics, Heisenberg's Uncertainty Theorem, etc.).

A quantum-based hardware generator is actually practical. Examples that have been used are:

- The interval between the emission of particles during radioactive decays. This technique is used at a web site to generate random bits that are available on-line from the decay of Krypton-85.[15] This source generates only 30 bytes per second and requires a cumbersome (and dangerous?) pile of hardware.
- The thermal noise across a semiconductor diode or resistor. This is the approach most often used in add-on PC hardware including the Intel RNG.
- The frequency instabilities of multiple free-running oscillators. This approach is the basis of the VIA C3 Nehemiah RNG approach (see Chapter 5). While it is implemented dif-

[15] http://www.fourmilab.ch/hotbits/

ferently than the resistor-based approach, ultimately, the source of randomness is the same.

- The charge developed on a capacitor during a particular time period.

These sources have been used in a few commercially available add-on random number generator devices, none of which have achieved much visibility or use. Since they are peripheral devices such as PCI cards and serial port devices, these commercial hardware generators are expensive and cumbersome. In addition, they are all made by universities or small companies with limited marketing and distribution capability. Some examples of such add-on hardware generators are found in this reference.[16]

There is one notable exception to the add-on strategy. Intel incorporated a hardware RNG in an optional chip in a particular chipset family. Since this is the only comparable implementation to the VIA C3 Nehemiah processor RNG, this Intel device is covered subsequently in additional detail.

2.4 HOW TO VERIFY RANDOMNESS

Given the differences in generation algorithms and the inherent theoretical problems with software generators, the obvious question is "how do we compare or quantify random number generator 'goodness'?" For any serious application or generator design, it is essential to have tests that *quantify* randomness.

In one sense, the definition of a random number is simple—the probability that each bit generated is a one (or zero) should be ½. Unfortunately, that definition doesn't help us directly since we can't actually test the *probability* for the next bit to be generated. All we can do is to study bits that are actually generated and analyze their characteristics to see how likely it is that the generator is random based on what it has generated. This approach turns out to be complicated, and in fact, there is no single or simple set of tests that can accurately identify randomness.

The underlying difficulty here is that truly random numbers occur randomly. That is, every possible value can be generated by a truly random source. If we sample 64 bits, for example, from a truly random source, there is a tiny but non-zero probability that all 64 bits will be zero. In that case, should we reject the source as not random? A potential answer is to collect many 64-bit samples and check to see how many times all zeros appears. Statistically, we should expect one all-zero set per 2^{64} samples. It is usually impractical to collect such a comprehensive set of samples, however. Indeed, because all outputs should be equally unlikely, no particular output provides "proof" of non-randomness. As a result, even the appearance of multiple all-zero sets would not prove that the source is nonrandom. Conversely, what if the sample values are 0, 1, 2, 3, 4, etc.? The values and bits seem to be appearing in the right statistical frequency, but do we really believe that the source is random?

Solutions to this fundamental problem are complex and rely on some understanding of statistical concepts. This testing topic is very important, however, and requires some attention in order to determine the proper use of any RNG, including the VIA C3 Nehemiah processor RNG.

[16] http://faust.irb.hr/~stipy/random/hg324.html
http://comscire.com/index.htm
http://mywebpages.comcast.net/orb/
http://www.protego.se/sg100_en.htm

Rather than provide a summary of the testing approaches here, our summary background is included (along with some results from the VIA C3 Nehemiah RNG feature) in Chapter 7.

2.5 THE INTEL 82802 RNG

Intel provided a hardware random number generator in the 82802 Firmware Hub. This chip is an optional component in the Intel 8xx series chipsets using flash technology to store and manage system and video BIOS.[17] It appears that Intel has discontinued this chip and it is no longer available in modern systems.

A good summary and analysis of this Intel RNG hardware capability is found in an Intel-commissioned paper written by Cryptography Research, Inc., a cryptography consulting company.[18] CRI has also been commissioned to evaluate the VIA C3 Nehemiah processor RNG.

The VIA C3 Nehemiah Processor RNG was specifically designed to provide significant advantages over the Intel implementation. The advantages arise from two important philosophical differences in the design approaches:

- Intel added their RNG feature in a peripheral chip. The VIA C3 Nehemiah incorporates the RNG in the processor itself.
- Intel assumed that software would access random numbers through a device driver (provided by them), where they could add or constrain functionality. VIA assumed that the application or operating system would want to have *direct* access to the hardware-generated random numbers.

Table 1 summarizes the most important features of both RNG devices.

[17] Information about the Intel 82802 FWH can be found at
http://www.intel.com/design/security/rng/rng.htm (general information)
http://developer.intel.com/design/chipsets/datashts/290658.htm?iid=ipp_815chpst (the datasheet)
http://www.intel.com/design/chipsets/manuals/298029.pdf (the programmers reference annual)

[18] Benjamin Jun et al., *The Intel Random Number Generator,* Cryptography Research, Inc., 1999
http://www.cryptography.com/resources/whitepapers/IntelRNG.pdf

Table 1. Summary Features of the Intel RNG & the VIA C3 Nehemiah RNG

Feature	Intel RNG	VIA C3 Nehemiah RNG
Random Number Source	Electrical noise & pseudo-random environmental effects	Electrical noise & pseudo-random environmental effects (using a different design than Intel's)
Location	Within the optional 82808 Firmware Hub	Within the processor
Direct application access?	A protected mode device driver is always required	A new x86 instruction is provided to directly read the generated numbers. No device driver is required
Supports Hardware Multithreading?	No, reading RNG must be a "critical section"	Yes, reading RNG is an atomic instruction
Asynchronous Hardware Data Collection	Up to 1 byte	Up to 32 bytes
Generation Speed (throughput)	Variable… 13 μs/bit to \| 76–20K 50 μs/bit \|[19] bit/sec 75 Kbits/sec[20]	Variable… 600K–6M bits/sec
User Controllable Configuration	Not from hardware. The Intel device driver imposes a particular hash algorithm on the hardware generated values.	Yes, 4 levels of hardware control
Testing approach	Intel's *software* drivers perform a FIPS 140 test at startup	The operating system and applications can perform whatever tests they deem necessary
Miscellaneous Features… Programmable bit string filter User controllable DC bias Raw generated bits available	 No No Partial	 Yes, up to 64 bits Yes Yes

[19] From the Intel 82802 Firmware Hub Datasheet

[20] From Benjamin Jun et al.,

CHAPTER

3

PROGRAMMING INTERFACE

This chapter describes in detail the visible registers and instructions of the VIA C3 Nehemiah processor family RNG feature. Not all of these detailed functions are required for application usage of the RNG.

3.1 GENERAL ARCHITECTURE

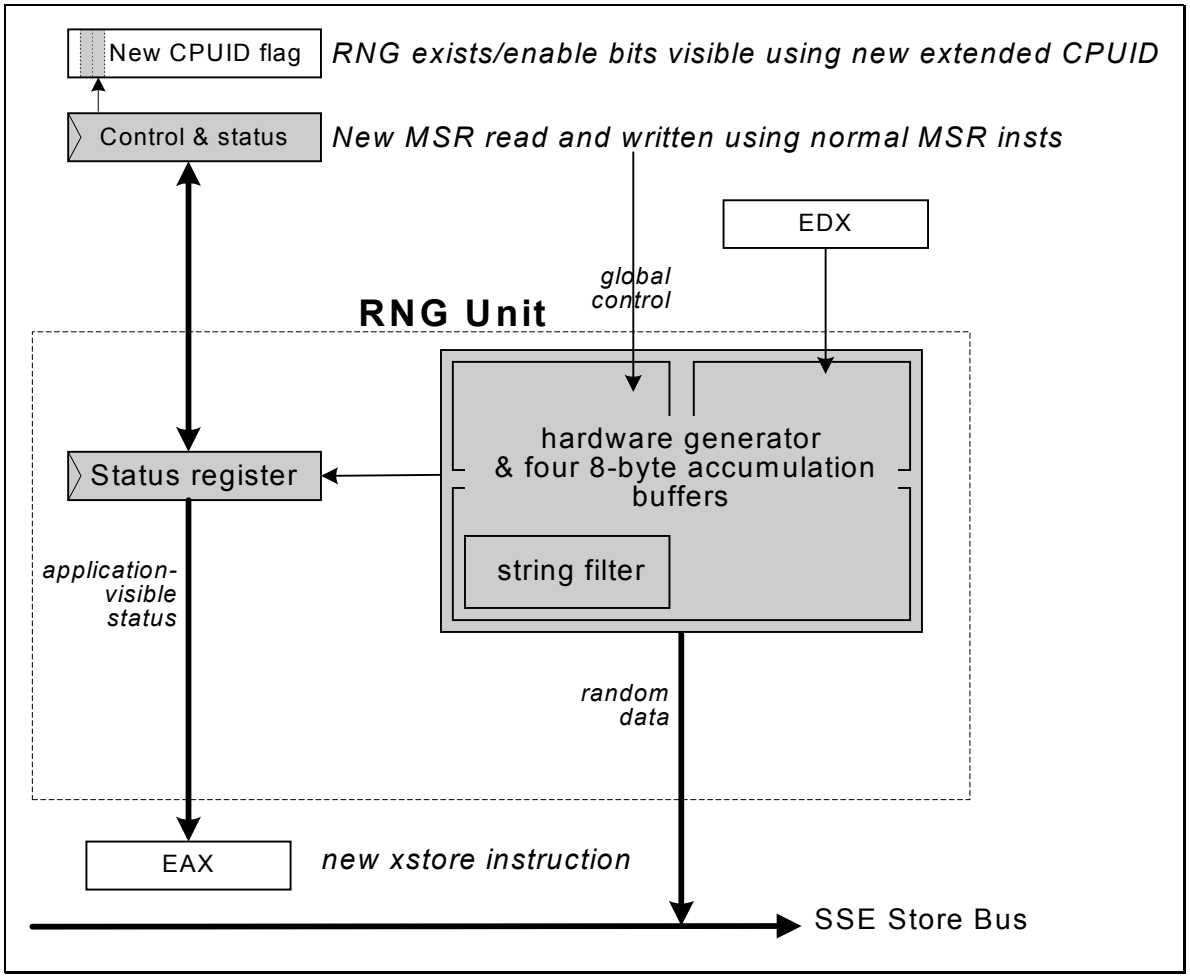
Figure 1 is a simple conceptual view of the visible architecture of the VIA C3 Nehemiah processor RNG feature. The major architecture components are:

- Two bits in the *Centaur extended* feature flags returned by the CPUID instruction. These bits identify (1) whether the RNG feature physically exists on the processor, and (2) whether it is currently enabled.
- A new RNG status and control Machine Specific Register (MSR). In addition to providing status information, this privileged register controls global options (they affect all applications) of the RNG.
- A high-speed hardware mechanism for generating the random data. Some global test features are available for analyzing behavior of the generator.
- A hardware datapath mechanism that collects the generated bits into large buffers. A global option is provided that can eliminate abnormally long strings of bits (as defined by the user).
- A new x86 non-privileged instruction—`xstore`—which stores the collected random data to memory, using the SSE store bus, as well as placing a copy of the status and con-

trol MSR in EAX so that applications can verify that the RNG is configured acceptably. An option exists that allows the microcode to return only certain generated bits, thus reducing the effective generation rate but improving the randomness of the data.

- A REP string version of `xstore` which allows easy collection of long contiguous strings of random data allowing transparent interrupts.

Figure 1. Visible RNG Architecture



Subsequent sections in this chapter describe in detail these registers and instructions. There are a lot of minor details and optional functions provided that may obfuscate the programming procedure for obtaining random numbers, but the basic approach is actually simple:

1. If special configuration is required (e.g., for testing purposes), enable the RNG unit and set control values via the RNG MSR. This is a privileged operation and rarely needs to be done since all of the globally settable options are used only for special cases and testing, and the default state at RESET is that the RNG is enabled.

2. An application can use `xstore` at any time to store random data *and* the instantaneously-accurate count of how many random bytes (0, 1, 2, 4, or 8) have been stored. This is an atomic instruction and can be used "concurrently" by any number of active tasks (and the operating system). Note, however, that any global options set in step 1 affect all users.

3. Examine the number of bytes returned in step 2, and repeat step 2, if necessary, until the desired number of bytes has been obtained. Or…

4. Replace steps 2 and 3 with a REP xstore instruction specifying the exact number of bytes desired.

"Paranoid" (e.g., cryptographic) applications should be advised to verify that the RNG configuration is valid (or at least unchanged) after each xstore operation. The concern is that a different application running on the CPU could modify the RNG configuration. If this is the case, then these "paranoid" applications should not use the rep prefix.

3.2 IDENTIFICATION & ENABLING THE RNG

3.2.1 OVERVIEW OF ENABLING

To use the VIA C3 Nehemiah processor RNG, three conditions must be met:

1. The RNG feature must physically exist on the chip. Its existence can be discovered from the Centaur Extended CPUID Feature Flags. This condition is set as part of the manufacturing process and cannot be changed by software.

2. The RNG feature must be enabled and configured appropriately. The enabled state can be discovered from the Centaur Extended CPUID Feature Flags. The RNG feature can be enabled/disabled by a writing to a new MSR. The RESET default is (usually) enabled. But, see the next condition.

3. SSE instructions must be enabled via the standard x86 method of enabling the FXSAVE/FXRSTOR instructions using CR4[9] This CR4 enabling also enables the full set of SSE instructions on both Intel and VIA C3 Nehemiah processors. If this CR4 bit is not set, the RNG behaves the same as if it were disabled via the RNG MSR, regardless of the setting of the RNG enable bit in the RNG MSR. We assume that a reputable operating system enables SSE instructions (if they exist) as part of its normal boot process. The operating system (or BIOS) must make sure that SSE is enabled before it can use the RNG.

> **Implementation Note:** This dependency on SSE is because the RNG datapath just happens to be buried inside the SSE datapath. This is a implementation choice, future VIA processors may eliminate this dependency.

3.2.2 CPUID INSTRUCTION IDENTIFICATION

The VIA C3 Nehemiah (Family=6, Model=9, Stepping>=3) processor supports the Centaur Extended Feature Flags. When the CPUID instruction is executed with EAX=0xC0000000, the C3 Nehemiah returns 0xC0000001 in EAX. Note that earlier VIA C3 processors do not support the Centaur Extended CPUID Functions and will return EAX=0xC0000000 or

EAX=0x00000000 (see the VIA C3 Nehemiah Processor datasheet for details on the extended CPUID function).

If a CPUID with EAX=0xC0000000 returns a value in EAX >= 0xC0000001 then Centaur Extended Feature Flags are supported. A CPUID with EAX=0xC0000001 then returns the Centaur Extended Feature Flags in EDX. There are two bits in EDX that describe the RNG:

- EDX[2] 0=RNG feature does not exist on this chip. A RDMSR/WRMSR of the new MSR 0x110B will cause a General Protection Fault, and the new `xstore` instruction will always cause an Invalid Opcode Fault.

 1=RNG feature exists, a RDMSR/WRMSR of MSR 0x110B is allowed. The behavior of `xstore` is dependent upon whether the RNG is enabled or not.

- EDX[3] 0=RNG is disabled. The new `xstore` instruction will cause an Invalid Opcode Fault. This bit reflects the result of the setting of bit 6 of MSR 0x110B.

 1=RNG is enabled (usually the RESET default). This bit reflects the result of the setting of bit 6 of MSR 0x110B. If 1, the `xstore` instruction can be used at any privilege level (provided CR4[9] =1)

3.2.3 MSR 0x110B

A new MSR—0x110B—provides both status and control of the RNG feature. In addition, a copy of the MSR contents as they are at the start of execution of an `xstore` instruction is placed in EAX as a result of executing an `xstore` instruction. (As specified in section 3.5.1

The EDX portion of this MSR is undefined. The EAX portion of the MSR contains:

31:22	21:16	15:15	14:14	13:13	12:10	9:7	6:6	5:5	4:0
Reserved	String filter count	String filter failed	String filter enable	Raw bits enable	DC bias	*Reserved*	RNG enable	*Reserved*	Current byte count ***read-only***

Reserved bits have undefined and unpredictable values. In the following detailed description, "O" means output by the processor, "I" means the field can written by software. In addition, the phrase "when the MSR is read" means when it is read by a RDMSR instruction or when it is copied into EAX as part of an `xstore` instruction execution.

VIA recommends that when writing the MSR, all reserved bits should be set to zero. It is not possible to misconfigure the current version of the VIA C3 Nehemiah RNG, but subsequent versions may define these reserved bits and software that improperly detects the processor stepping may misconfigure these later steppings.

The detailed fields are:

- **Bits 4-0: Current Random Byte Count (O).** When the MSR is read, this *read-only* field contains the exact number of random bytes that are "currently" available for storing. Due to the asynchronous generation of random bytes, and the fact that multiple programs may be storing data using `xstore`, the number that appears may not be the same as the number stored on the next execution of `xstore`.

 When this field appears in the copy of the MSR placed in EAX by the execution of `xstore`, the number is the exact number of bytes actually stored by the `xstore`.

- **Bit 6: RNG Enable (I).** When set to 1, this bit enables the RNG functions. When set to 0, the new `xstore` instruction becomes an invalid. The extended CPUID feature flag that indicates the RNG is enabled (EDX[3]) is a copy of this bit.

 This enable bit may be set internally by the RESET process. In this case, the effect on the processor is the same as if a write MSR set the enable bit.

- **Bits 12-10: DC Bias (I).** These bits control the DC bias supplied to the random number generator (see chapter 5). These bits may affect the speed of the generator and the randomness of the generated bits.

 The ability to change this field is provided primarily for test purposes *and should not be changed during normal use*. See section 4.4 for more specifics.

- **Bit 13: Raw Bits Enabled (I).** If this bit is set to 1, the "von Neumann compressor" (or *whitener*) in the hardware generator is disabled and the raw bits produced by the random generator are delivered to the accumulation buffers (see the hardware summary in chapter 5). A 0 in this bit selects the whitener function, which is required for producing truly random values. If it is desirable to sample the raw bits, then the string filter must also be disabled (bit 14) since the raw bits generated go through the string filter mechanism.

 The ability to change this field is provided only for test purposes *and should not be set during normal use*.

- **Bit 14: String Filter Enable (I).** If set to 1, this enables a test feature that filters out strings of contiguous ones or zeroes longer than the value set in bits 21:16. The exact behavior of this function is defined in section 3.4.1. Note that enabling the string filter will introduce non-uniform statistical characteristics in the output.

 The ability to change this field is provided only for test purposes *and should not be set during normal use.*

- **Bit 15: String Filter Fail (I/O).** This bit indicates that while the string filter was enabled, the hardware detected a string of contiguous ones or zeroes longer than the value defined by the filter count in bits 21:16. The hardware actions are defined in section 3.4.1. Only the hardware can set this bit to a 1, but a MSR write can set it to 0 to clear it.

- **Bits 21-16: String Filter Count (I)**. The value in this field defines the bit-string length (8–63) to be used by the string filter. *Bit string lengths less than 8 will produce unexpected results and should not be used.*

3.3 NEW X86 XSTORE INSTRUCTION

The VIA C3 Nehemiah processor RNG feature implements a new x86 instruction: `xstore`. This instruction is enabled by the RNG enable bit in MSR 0x110B; an Invalid Instruction exception occurs if `xstore` is executed when not enabled. The instruction formats and general behaviors are described here, the specific functions performed by these instructions are described in the subsequent section.

3.3.1 XSTORE: STORE RANDOM DATA INSTRUCTION

This instruction stores zero, four, or eight random bytes in memory and places a copy of the resulting RNG MSR 0x110B in EAX.

The opcode of `xstore` is an invalid opcode in normal x86 processors. When enabled, the format of the `xstore` instruction is:

0x0F	0xA7	0xC0

Any values other than 0xC0 in the last byte of the instruction will cause undefined behavior.

The `xstore` instruction requires that:

- ES:EDI points to the starting (low-order) memory address where the random data is to be stored. No segment override is possible.
- EDX specifies the rate at which the random data are returned, and the corresponding level of randomness. Only the lower two bits are meaningful, and the upper 30 bits will be set to zero by the instruction. The meaning and effect of the value in EDX are discussed subsequently.

The `xstore` instruction has many similarities with the x86 `stos` instruction:

- The value in EDI is updated for the number of bytes stored.
- An address-size prefix affects the size of EDI used and updated.
- A REP prefix affects the behavior of the `xstore` instruction (see the next section).
- The usual address exceptions (past the segment limit, etc.) can occur.

In contrast to `stos`, however, the `xstore` instruction performs differently in some areas:

- A REPNE prefix causes an Invalid Instruction exception.
- In addition to storing the random bytes, the status word is placed in EAX.
- An operand size prefix causes an Invalid Instruction exception.
- The specification of a 16-bit mode in CS is ignored; the full 32-bit EAX is always updated.
- The DF (direction flag) in EFLAGS has no effect: instruction execution always adds the number of random bytes stored to EDI.
- Either zero, four or eight bytes are always stored to memory depending on the data rate specified by EDX and the status of the RNG hardware (see section 3.3.2 below). Software must provide always an appropriate size writable area at the ES:EDI destination address.
- Zero, one, or two 4-byte stores perform the store. Thus, all memory-operand restrictions and exceptions that are applicable to a eight- or four-byte store are also applicable to the `xstore` instruction. If alignment check is enabled and is to be avoided, then the address in the starting DS:EDI should be 4-byte aligned. Similarly, segment limit checks, protection check, page faults, etc. behave as if the data size is four or eight bytes.

3.3.2 XSTORE: NUMBER OF BYTES STORED

Implementation Note: The RNG hardware internally provides multi-buffering using four internal eight-byte buffers. When the active eight-byte buffer fills with random unread data, the visible counter stops at eight, but incoming random bytes start to fill the next buffer starting at byte zero. When all eight bytes in the first buffer have been read, the buffers are switched: the count and the data in the next buffer become visible, and the first buffer is available for overflow filling. This deep buffering approach typically allows the RNG hardware to accumulate many bytes of random data completely overlapped with instruction execution.

The rules for the number of bytes stored by xstore are:

- The `xstore` instruction checks to see until an eight-byte hardware accumulation buffer is full. If fewer than 8 bytes are available, the instruction performs no store, and returns information with a zero byte count.
- If at least 8 bytes are available in the hardware buffers, the appropriate bits are selected (defined by EDX, more on this subsequently) and either one or two four-byte stores are performed as required.
- Note there may be less than four valid *random* bytes stored depending on EDX. In this case, the remaining bytes of the store contain zeroes. The byte count in EAX and the update to EDI are based on the number of *random* bytes stored, not on the size of the stores themselves.

Here is a summary:

EDX 1:0 value	Total bytes available	Total bytes stored	Random bytes stored	EAX[4:0] & EDI update byte count
doesn't matter	< 8	0	0	0
00	>=8	8	8	8
01	>=8	4	4	4
10	>=8	4	2	2
11	>=8	4	1	1

3.3.3 REP XSTORE INSTRUCTION

The `REP` prefix on an `xstore` instruction performs as follows:

- The ECX value defines the total number of random bytes to be stored.
- The `xstore` function is repeatedly performed until the ECX number of random bytes are stored.
- Zero, one or two 4-byte stores are performed as for each iteration of `xstore` but ECX and EDI are updated automatically such that random bytes are concatenated into a contiguous string and iterations continue until the desired number of *random* bytes are stored.

- The implications of storing fewer random bytes than the size of the store are that unaligned four-byte stores may be performed. These are slower than aligned stores and and will cause an alignment check if it is enabled.
- Another implication is that on the last iteration, `REP xstore` may store up to seven bytes to memory beyond the last random byte stored. Adequate memory must be allocated based on the EDX value to contain the last one or two four-byte stores.
- The `REP` execution is interruptible. When an exception or interrupt occurs and is taken, or the instruction completes execution, (1) ES:EDI is updated to address the next location for storing *random* byes, (2) ECX is updated to contain the number of random bytes remaining to be stored, and (3) EAX contains a copy of the current status word. *However, the byte count field (4:0) in the status word is not defined for the REP case.*
- Code run during an interrupt could, at least in theory, modify the RNG configuration. In (unusual) environments where this may occur, application programmers should be aware that the RNG status reported in EAX at the end of the operation may not reflect the configuration at the beginning of the operation. If it is important to make sure that the RNG configuration does not change, the non-REP `xstore` should be used.
- The same rules apply as for a non-REP `xstore` for the REPNE prefix, the AS prefix, the OS prefix, the size for alignment purposes, the size for other exceptions, etc.

Programming Note: There is a theoretical problem with using `REP xstore` instruction because (1) the instruction doesn't return control until all requested bytes are stored, and (2) the generation bit rate is variable. As discussed in section 4.3, the `REP` version of `xstore` prevents easy detection of a situation where the RNG hardware is not returning any bytes. In practice, this not a real risk, since this situation cannot occur without a catastrophic hardware failure. Applications can address this possibility by performing a startup test as recommended in section 4.1.

3.4 STRING FILTER FUNCTION

3.4.1 BASIC BEHAVIOR

The optional string filter feature is globally enabled via MSR 0x110B. The details of this function are a little messy, but the bottom line is simple: if enabled, this feature suppresses certain runs of contiguous ones or zeroes beyond a specified limit. *This mode is documented for completeness, but should not normally be used. The feature may be removed in future versions.* The detailed string-limit algorithm is:

- The hardware maintains a counter of contiguous bits and a flag of whether the count is for ones or zeroes (the "current type").
- When a new random byte is ready to be added to the internal hardware buffers, the number of leading ones or zeroes is calculated (this may be the entire byte). If all bits of the byte are the same as the current type, this leading number is added to the count.
- If this add causes the counter to exceed the string limit count, *and if the string filter function is enabled,* the incoming byte is discarded and the "fail" status bit is set in the control word. Random byte collection continues with the next byte to be delivered.

- If the byte is not discarded, and if the byte's bits are not entirely the same as the current type, then the trailing bit count and type is placed into the count and current type flag. That is, a new string count is initiated.
- If the byte is entirely the same as the current type nothing special happens, the byte has been accepted and the count increased by 8 and the count has not exceeded the string limit count.
- The string limit count must be greater than seven or the results are undefined, and in fact the RNG hardware may fail to return data.

This algorithm can ensure that no more than *n* contiguous ones or zeros is ever delivered to the random number accumulation buffer. Note that if EDX[1:0] is non-zero then an `xstore` or `REP xstore` stores less than every bit from the accumulation buffer which may result in more than *n* contiguous ones or zeros actually stored to memory. In other words, the *whitener* (if enabled) and the string filter (if enabled) are located prior to the accumulation buffer, whereas the data rate filter specified in EDX[1:0] is located after the accumulation buffer. For an architectural diagram, see Figure 2. RNG Hardware Overview, in chapter 5.

Warning: The string filter can significantly alter the statistical characteristics of the output data. In general, it is strongly recommended that this feature only be used for testing purposes. If applications require post-processing of RNG output, this is usually best to implement in software.

Important Curiosity: The string filter considers the least-significant bit (lsb) of a byte stored at address *A* to be adjacent to the most-significant bit (msb) of the next byte (stored at address *A+1*). This is equivalent, from a probability point of view, to considering the msb of a byte at address *A* to be adjacent to the lsb of the next byte (stored at address *A+1*). If these two bytes are loaded into a register as a single word, however, then the msb of the byte at address *A* will be adjacent to the lsb of the next byte. This may result in more than *n* contiguous ones or zeros in the register, even though the hardware may not have indicated a string filter failure.

3.4.2 MULTITASKING ISSUES

The string filter operates on the hardware-generated bytes, independent of multi-task considerations. The generated stream of bytes crosses a task switch boundary such that the stream of random bytes is split between multiple tasks. In this case, the string filter fail may occur even though any particular task might not see a string that exceeded the limit. Conversely, any particular task may see a long string of ones or zeroes without a hardware string filter failure.

Important Curiosity: As discussed in above, it is possible that bits that are adjacent in the same byte in memory were not adjacent when the string filter inspected them. This is due to (1) the bit selection mechanism of `xstore` and `REP xstore`, and (2) the funny way bytes are related to each other. Multi-threading introduces yet another opportunity for bytes that were not adjacent when the string filter inspected them to end up adjacent in a particular thread (since a task switch may have removed intervening bytes).

3.5 STARTUP STATE & ACTIONS

3.5.1 RESET DEFAULTS

As part of the RESET process, the following RNG actions are automatically performed *if the RNG unit is present:*

- MSR 0x110B is initialized with the following value: 0x00000040
- This enables the RNG unit and sets the default values of all control fields to 0:
 string filter disabled
 raw bits disabled (normal whitener function enabled)
 DC bias = 0
- The RNG hardware accumulation registers are cleared to 0 and the accumulation count set to zero
- Random numbers start to accumulate overlapped with any other RESET or software function.

3.5.2 INIT ACTIONS

As opposed to RESET, the INIT signal causes no *direct* actions relative to the RNG feature. That is, INIT does not internally cause a write to MSR 0x110B. The INIT signal causes CR4, however, to be reset. Since this resets the FXSAVE/FXRSTOR enable, CR4[9], the RNG is indirectly disabled.

This behavior should present no special problem since any reputable operating system will re-initialize CR4 following an INIT.

3.5.3 RNG ENABLE

Writing MSR 0x110B with the enable bit (bit 6) set to 1 causes the following actions:

- The `xstore` instruction is enabled: this opcode no longer causes an Invalid Instruction exception.
- The internal hardware buffers are cleared to zero.
- The internal accumulation count in the string filter is reset to zero.
- The various options defined by the MSR (string filter, raw bits, DC bias) are set to the value written to the MSR.
- The random number generator is internally enabled and random bits start accumulating into the hardware buffers. This accumulation is triggered solely by enabling the RNG feature and is independent of the `xstore` instruction.

3.5.4 RNG DISABLE

Writing MSR 0x110B with the enable bit (bit 6) set to 0 causes the following actions:

- The `xstore` instruction is disabled: this opcode now causes an Invalid Instruction exception.
- The internal hardware buffers are cleared to zero.

- The internal accumulation count in the string filter is reset to zero.
- The random number generator is internally disabled to reduce power consumption (see section 6.3).

3.5.5 WRITING THE MSR

A write to MSR 0x110B that *changes any* of the writable fields causes *a "load reset"* of the RNG hardware. The load reset actions are

- The internal hardware buffers are cleared to zero.
- The internal accumulation count in the string filter is reset to zero.
- The various options defined by the MSR (string filter, raw bits, DC bias) are set to the value written to the MSR.

This load reset does not, per se, disable the random generator; random values continue to be generated and accumulated unless the disable bit (bit 6) is set to 0 by the MSR write.

3.6 MANUFACTURING TESTS

The RNG feature on every part is tested as part of manufacturing test. While it is impossible to perform a strenuous statistical test (see chapter 7 for more discussion on testing), the most important test from the government standard FIPS-140 is run.

A FIPS 140-2 monobits tests is executed (20,000 bits) at EDX=0 (the lowest quality setting). Passing this test verifies that the bit generator and the associated datapath are alive and working with no stuck bits, etc. It does not assure that the generator is statistically good enough to be considered truly random, but will generally detect catastrophic failures that result in highly biased or stuck output. (Actually, a failure on this FIPS test does not really mean that the generator is not random, but since the probability of a failure on one sample from a truly random generator is so low, the presumption is that a fail means broken.)

Important: *At this time,* if a processor fails the RNG test, or times out after a reasonable time without generating the expected number of bits, the random generator is permanently disabled for that particular part. This means that the RNG feature may be marked nonexistent on some VIA processors. VIA may change the RNG manufacturing tests at any time. Customers requiring RNG support may wish to contact VIA to ensure that all parts ordered have active RNGs.

CHAPTER

4

PROGRAMMING CONSIDERATIONS

This chapter discusses various programming topics that are of special interest in using the VIA C3 Nehemiah RNG feature.

4.1 RANDOMNESS CONTROL

EDX controls the selection of which hardware-generated bits are returned by the `xstore` instruction. The higher the value, the slower the effective bit rate, and the better the randomness characteristics of the values returned. Chapter 7 contains some test data showing this randomness effect. The obvious question is "how should an application set EDX?"

We suspect that most applications will want the best randomness available with the least effort. Since the hardware speed is very fast even with the EDX=3, these applications should use EDX=3. This option provides very good randomness with no additional software work required at, typically, a fast 800K bits/second.

There may be applications, however, that need even higher bit performance. They may be satisfied with reduced randomness from the hardware. Alternatively, they may choose to use a good cryptographic generator (such as one based on SHA-1) seeded using the hardware RNG. The combination of lower EDX values and a good hash should be able to produce high quality values even faster than the EDX=3 hardware option (the hash should be able to keep up with the bit generation rate). The tradeoff, of course, is that a *significant* number of processor cycles are required to compute the SHA at these high data rates and technically the output ceases to be *truly* random.

In addition to the data in chapter 7, Centaur may be able to provide some advice or assistance to customers with unusual requirements based on our experience with various settings.

4.2 STARTUP TESTING

For some FIPS 140-defined security levels, the standard requires that a single FIPS-140 random number test be performed at startup time for a certified device. The need for some startup test is further indicated by the fact that the RNG feature is not fully verified statistically in our manufacturing process (section 3.6). Even if the product is not a NIST-certified device, *some* start-up test is a good idea in all environments. It is also a good idea to perform some test within an application before any collected random data is trusted.

Which startup test should be run is not obvious, however. There are two basic things that a startup test could be testing for:

- Whether the RNG feature is basically working or not (for example, does it have a stuck bit such that no data is ever returned).
- The randomness characteristics of the numbers generated.

The monobit test is performed as part of the manufacturing process (see Section 3-6). For many applications, a FIPS-140 test is appropriate (see chapter 7 for more information on the tests referenced in this section). For randomness testing, however, the FIPS test is not very stressful. In addition, FIPS-140-1 and FIPS-140-2 set different pass/fail limits (plus, the FIPS-2 limits for two tests are poorly chosen as they will cause even a perfect RNG to fail a significant percentage of the time). Furthermore, one sample of FIPS (20,000 bits) does not indicate much about the randomness. All it can do is say whether the RNG is (1) probably broken or (2) probably not broken. (If one sample fails "badly", you should assume the RNG is broken.) In summary, a single FIPS test cannot indicate whether the random generator is generating good random numbers. To verify the randomness correctly requires testing many 20,000-bit samples (and maybe more strenuous tests, see chapter 7).

Historically, performing a startup test has been a problem in some environments because of the elapsed time involved. Due to the high performance of the VIA C3 Nehemiah processor RNG, however, a FIPS test, for example, can be executed in about 25 milliseconds. (The execution time can be overlapped with the bit collection such that the bit collection time effectively determines performance.). This makes it not only feasible to run a startup test, but may make it feasible to test multiple samples at startup.

The next section describes a timeout potential when using the `REP xstore` instruction. Since a timeout is one of the problems that a startup test is intended to detect, the startup test should not use `REP xstore`.

As an assist to operating system and application developers, Centaur technology will provide sample assembler code that performs each FIPS 140 test (including timeout detection).

4.3 TIME-OUT ISSUES

In normal operation (the whitener enabled), it is theoretically possible that the RNG generates no new bits for a "long" time. This theoretical delay has no guaranteed maximum, even in a

correctly working part, in the same way that there is no limit to the number of sequential coin tosses that can come up heads. In practice, however, the likelihood of a long delay is negligible. This belief is based on (1) general statistical principles, (2) the inherent design of the random bit generator, and (3) our observations based on extensive testing. Our testing has never showed any deviation from the basic generation rate over long periods of operation (many weeks). The bit rate seems to be very constant from startup to shutdown. If startup tests run successfully (i.e., the hardware has not failed), applications should not need to detect timeouts.

Of course, a very conservative application can easily add a "time-out" check, since the `xstore` instruction is usually used in a loop counting the bytes stored. It is easy to add another count that can detect n `xstore` instructions with no data returned. Alternatively, a time limit can easily be set and checked by the application.

The timeout detection using the `xstore` instruction works because `xstore` always immediately returns to execute the next instruction even if no random data is available. The `REP xstore` instruction, however, does not return control until the random byte count specified in ECX is stored. There is no built-in timeout check—if the RNG has died, the `REP xstore` will wait forever. Since the `REP xstore` is interruptible, however, this case hangs the application, but not the operating system or other applications. It is also theoretically possible to detect a `REP xstore` timeout by intercepting an interrupt and examining the ECX count compared to some timer information about how long the `REP xstore` has been executing. This is very tricky and requires hooks into the operating system, so we recommend the following: do not use the `REP xstore` instruction if you are concerned about a RNG timeout condition.

4.4 DC BIAS

The DC bias adjustment is primarily intended for Centaur Technology's internal analysis, test, and debug. Our philosophy, however, is to provide all existing functions to software in case there is some external value in them. Following that philosophy, we make the DC bias control available.

The VIA C3 Nehemiah processor RNG feature usually performs best with the default DC bias setting (0). It is possible, however, that alternate DC bias settings may improve bit-generation speed, or, if the whitener is disabled, improve randomness characteristics. The sophisticated or curious user may want to experiment with changing the DC bias. This requires protected-mode access (for writing the RNG MSR).

One might think that a boot-time characterization could be used to determine the optimal DC bias setting for a particular part. The problem with this approach is that an *extensive* number of samples must be collected and analyzed statistically (using the techniques summarized in Chapter 7). The time required for a comprehensive determination of the correct DC bias is probably prohibitive for the boot process. Setting a non-standard DC bias setting may also cause other applications to conclude that the RNG is mis-configured. It may be feasible for certain applications, however.

For those serious users interested in optimizing the DC bias, Centaur Technology may be able to provide some advice or assistance based on our testing experience.

CHAPTER

5

SUMMARY HARDWARE DESCRIPTION

This chapter summarizes the internal implementation of the random number hardware. This information is not needed for a program to use the RNG feature, but it may be of interest since the design of random generator is critical to producing true random numbers.

5.1 RANDOM BIT GENERATOR

Figure 2 provides a pictorial overview of the RNG feature. The basic components and functions of the bit generator are:

- There are three fast oscillators, and one slow oscillator. These are free running ring oscillators; that is, their frequencies are not synchronized or stabilized and individually drift over time. The fast oscillator frequencies vary around a center of about 1 ns. The slow oscillator varies around a center of about 20 ns. These oscillators are laid-out at right angles to each other to accentuate silicon differences.

 These oscillator frequencies are influenced by many factors:

 - Random quantum effects of the transistor behavior: thermal noise in the channel, gate leakage caused by tunneling, etc.
 - Dynamic voltage variations
 - Dynamic temperature variations
 - Dynamic electromagnetic effects

 - Process differences in silicon dimensions. The oscillators are laid-out at right angles to each other to further accentuate silicon differences.
 - Process differences in silicon characteristics such as implant characteristics
- The operating voltage of all oscillators is partially controlled by a single "DC bias" which can be adjusted in eight steps via a programmable value.
- The output of two of the fast oscillators is XOR'd to produce a variable signal.
- This variable signal further (in addition to the DC bias) modulates the voltage bias of the slow oscillator.
- The output of the variably biased slow oscillator is used as the clock to sample a bit from the third free-running fast oscillator.
- Each two bit pair generated is processed through a "von Neumann whitener", which may output a single bit or nothing. If a bit is produced, it is then accumulated in an eight-bit shift register. This whitener function will reduce or eliminate any residual bias in the hardware generator. For example, the whitener should correct for biases in the third oscillator's duty cycle. For test purposes, this whitener function can be bypassed, but this is likely to introduce significant biases in the RNG output.

 The input/output function of the whitener is

Input	Output
00	Nothing
01	0
10	1
11	Nothing

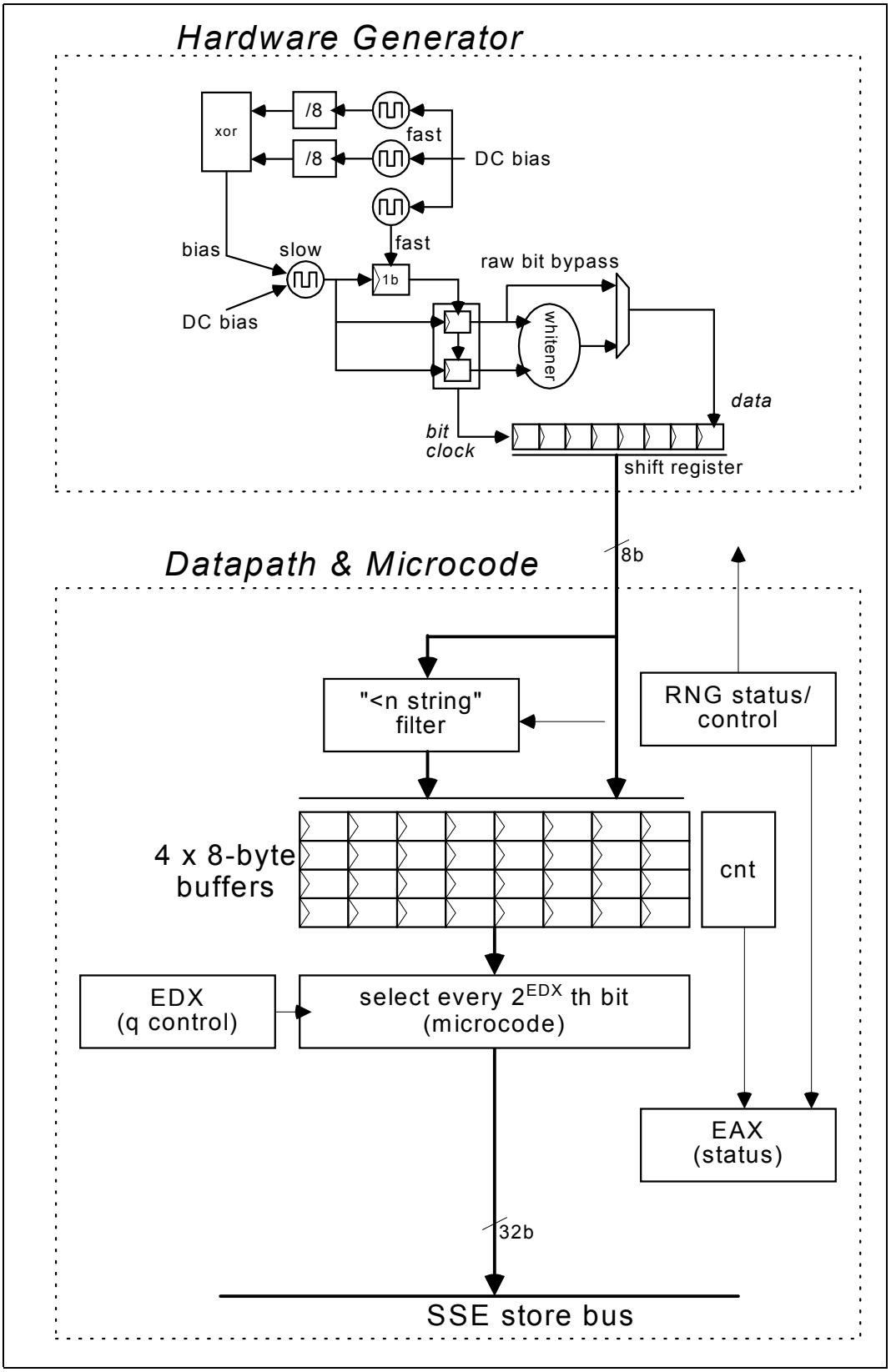
- When eight new bits have been accumulated, a strobe causes the eight bits to be grabbed by the RNG data path (which is running using the synchronous processor clock).

5.2 DATAPATH & MICROCODE FUNCTIONS

Once eight bits have been generated they are delivered to the synchronous datapath:

- Each incoming byte is examined by the string filter function. If it is enabled, it may suppress the delivery of the byte to the accumulation buffers. The string filter function is further described in Section 3.4.1
- The incoming bytes are placed sequentially in the next consecutive position of four 8-byte buffers. The hardware maintains FIFO pointers and a count so that it can properly store the incoming bytes and access the requested bytes.
- The `xstore` instruction itself is implemented in microcode. When executed, the microcode checks to see if eight new bytes are available in one of the buffers. If not, it returns with a zero byte count indicated. When all eight are available, the microcode performs the one-of-n selection (specified by EDX), assembles the resulting data, and performs the store.

Figure 2. RNG Hardware Overview



CHAPTER

6

PERFORMANCE & POWER

This chapter provides performance and power consumption data for the VIA C3 Nehemiah RNG feature.

6.1 RANDOM BIT GENERATION SPEED

The effective rate of generating random bits is variable and depends upon the processor's chip voltage, its temperature, process variations, phase of the moon, and so forth. Based on nominal *simulated* conditions, performance is as follows:s

- A new "raw" random bit is generated approximately every 20 nanoseconds (ns). This rate can vary significantly based on the DC bias, the process parameters, the voltage, etc.
- It takes at least two raw bits to generate a delivered bit (this assumes normal operations where the whitener is enabled).
- Two out of the four 2-bit possibilities will generate a delivered bit. If the raw bits are statistically equally distributed, then the whitener will produce an average of one output bit for every four input bits. The raw-bit distribution is typically biased, however. In this case, it may take many raw bits to generate a delivered bit. For estimating purposes in this section, we assume that it takes 8 raw bits to generate each delivered bit (as opposed to the four assuming a balanced distribution).
- We assume for this calculation that EDX=0; that is, every bit generated (with whitener) is delivered to software.
- Thus, a conservative theoretical bit rate for EDX=0 is:
 One delivered bit every 20 ns * 8 = 160 ns

Rounding down, we get a conservative bit rate of approximately 6 Mb/sec.

In practice, the actual rate may be faster or slower. In fact, in certain catastrophic error circumstances or if the RNG Is misconfigured, it may be possible that no bits are ever delivered. It is up to the application to detect any "time-out" conditions that may arise from such a variable rate.

Tests on sample C3 Nehemiah processors show, over a wide range of temperature and bias voltage settings, that bits are actually delivered to the accumulation buffers at an average rate of approximately 6 Mb/sec.

Tests have also shown that there is typically a minor correlation between sequential bits delivered to the accumulation buffers. To reduce this correlation, the `xstore` instruction uses the value in the EDX register (mod 4) as a divider, and returns only every 2^{EDX} bit of those produced by the generator. For more information, see section 7.

6.2 NEW INSTRUCTION PERFORMANCE

6.2.1 XSTORE INSTRUCTION

The `xstore` instruction is partly implemented in microcode. The number of execution clocks depends on whether random data are available, and on the value in EDX (the data rate register). The `xstore` does not return any random data unless the eight-byte accumulation buffer is full. If fewer than 8 bytes are available, the instruction returns status information regarding the RNG with a zero byte count. When an accumulation buffer is ready, `xstore` returns the bits in memory as pointed to by ES:EDI and in the quantity specified by EDX. The status information in EAX correctly indicates the actual number of bytes stored.

The following table gives estimates for the number of clocks required for the `xstore` instruction. For comparison, remember that on a 1 GHZ processor it requires an average of about 11,000 clocks to fill a single 8-byte accumulation buffer (1 GHz x 64 bits / 6Mb/sec).

Table 2. xstore Execution Clocks

EDX	Data Ready	Random-Data Stored	Execution Clocks
0-3	No	0	≈30
0	Yes	8	≈55
1	Yes	4	≈310
2	Yes	2	≈200
3	Yes	1	≈120

6.2.2 REP XSTORE INSTRUCTION

The `REP xstore` instruction is executed as an internal microcode loop of individual xstore instructions until the specified number of bytes have been collected. If the requested count is small and the data is available in the hardware buffers, the execution clocks of `REP` xstore is slightly more than shown for xstore in Table 2 .

For the more typical use of collecting a large number of bytes, the speed of `REP` xstore becomes effectively the bit speed plus a few hundred clocks. The bit generation speed dominates this number.

6.3 POWER MANAGEMENT

The RNG hardware is clock-enabled when the MSR enable bit is set. At this time, the random number bit generator is enabled. In addition, the RNG data path associated with accumulating the data, counting bytes, etc. is clock-enabled. Thus, dynamic power starts being consumed by the RNG unit when it is enabled (normally by default at RESET).

Whenever all four 8-byte accumulation buffers are full, the random number bit *generator* clocks are temporarily disabled. When a buffer location becomes free, the generator is re-enabled. The associated RNG data path clocks are never disabled, however, until the MSR enable is cleared. Even though data path clocks are not disabled, the power consumption of this logic is very small.

Relative to the SSE unit power management, the new `xstore` instruction behaves like SSE instructions. That is, when the `xstore` instruction is "seen" by the execution unit, the entire SSE unit is enabled until no further SSE instructions or `xstore` instructions are seen, at which point the SSE unit is disabled.

CHAPTER

7

RANDOMNESS TESTING

This chapter summarizes the background and most important tests for quantifying randomness. In addition, a summary of the randomness testing performed by Centaur is provided along with some sample results. Users, of course, may want to perform their own evaluations.

7.1 INTRODUCTION TO TESTING RANDOMNESS

7.1.1 GENERAL BACKGROUND

There is no simple way to answer to the question "is a particular generator random?" The only answers are statistical in nature and are only meaningful for relatively large sample sizes. We shall explore testing for randomness in some detail in this chapter, including a few equations. The intent is to provide adequate background to understand out randomness testing results In particular, this background section introduces technical concepts (*P-value* and α) that are extensively used in subsequent sections in this chapter.

While all randomness testing is statistical in nature, there are several different technical approaches. We primarily follow the approach proposed by the U.S government's National Bureau of Standards (NIST).[21] The essence of this approach is as follows:[22]

1. Identify *multiple* statistical characteristics of truly random numbers for which to test. No single statistical characteristic is adequate (for example, a sequence of 1, 2, 3, etc. satisfies a numeric distribution test, but obviously will fail a basic bit-to-bit correlation test). The basic theory here is that if something has tail feathers like a duck, walks like a duck, and quacks like a duck, then we may substitute it for a real duck. Unfortunately, there are many possible characteristics of a duck, and many possible statistical characteristics of random numbers. A particular generator may have a particular flaw that shows up on some characteristics, but looks random on others. Following the analogy, it may look just like a duck in all characteristics except for the webbing on its feet. For some applications (swimming), this difference may be catastrophic.

 Fortunately, a significant amount of research has been performed and documented relative to the desirable random number statistics. There are several sets of tests that been shown to be reasonably effective at identifying flaws in sources of randomness.

2. For each statistical test identified, perform statistical analysis to identify the probability that a truly random generator could have produced a sequence less random than the sample tested (for that particular test characteristic). This probability is called the P-value. Another way of stating this is that the P-value is the probability of obtaining a test statistic as large or larger than the one observed *if* the sequence is truly random.

 That is, the higher the P-value for a random number sequence, the more the sequence looks like it came from a random generator (for the particular test statistic). Hence, small P-values are interpreted as evidence that the sample is unlikely to come from a random source. For example, if the P-value for a particular sample is 0.3, it means that it is 30 % likely that this sample came from a random generator (relative to the particular test characteristic).

 This statistical analysis and resulting P-value calculation require serious mathematics, but the approach is well documented in the literature.

3. Identify the acceptable (to the application) probability that a test might reject a sample from a truly random generator. This acceptable probability is called the *level of significance*, or α. Typical values of α used in cryptography applications are from 0.01 to 0.001. An α of 0.005 means that one would accept that, on average, one sample out of every 200 samples from a truly random generator would be rejected by a particular test. The same α is generally used for all tests. Larger values for α increase the rate at which good RNGs will get rejected, while smaller values for α increase the likelihood that a bad RNG may pass the tests.

[21] Juan Soto, *Statistical testing of Random Number Generators,* NIST: 1999. From Proceedings of the 22nd National Information Systems Security Conference, 10/99. http://csrc.nist.gov/rng/nissc-paper.pdf.

[22] Andrew Rukhin, et al., *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications,* NIST Special Publication 800-22, 2001. http://csrc.nist.gov/rng/SP800-22b.pdf. This is a good description of the theory and practice behind random number testing, albeit quite technical.

4. The connection between of the calculated P-value and the target α is as follows: if the P-value $\geq \alpha$, then the sequence should be considered random (by the particular application for the particular test). A P-value $< \alpha$ would be considered a failed sample.
5. Perform all the tests on lots of *samples* (a contiguous sequence of generated bits). The result from a single sample (especially for the normal case of P-value $\geq \alpha$,) does not adequately identify whether a generator is truly random or not. The number of samples required depends on the test's requirements, but a careful analysis will typically include at least $1/\alpha$ number of samples, and often many more.
6. For the number of samples tested, calculate the statistical significance of the set of P-values across all samples and the failure rate. This analysis of the many samples finally gives us an overall confidence factor about the randomness of the generator.

An detailed example should make this process more clear. Let us consider the simplest test for randomness: the number of 1 and 0 bits in the sampled sequence. Let n be the number of sampled bits ($n > 1000$) and S be the absolute value of the difference between the number of 1 bits and the number of 0 bits. Calculate (we won't derive the mathematics here)

$$s = \frac{S}{\sqrt{n}}, \text{ and}$$

$$P - value = erfc\left(\frac{s}{\sqrt{2}}\right) \text{ where } erfc(z) = 1 - \frac{2}{\sqrt{\pi}}\int_0^z e^{-u^2}\,du$$

(The $erfc(z)$ functions is a commonly used statistical function called the *complementary error function.)* Solving these equations for a P-value = 0.0001, we find that the number of one bits (or zero bits), x, in a 20,000 bit sample must be between 9,725 and 10,275 (exclusive) in order for the resulting P-value to be ≥ 0.0001. That is, there is only a 0.0001 probability that a truly random sample of 20,000 bits will not have a value x such that $9{,}725 < x < 10{,}275$. That is, if our sample falls outside of these bit extremes, we could consider the sample not random with a 99.99 % confidence level.

Why did we pick a target of 0.0001? This was merely our choice as to how stringent our tests for random numbers should be. (Actually, this happens to be a specific example from an important government standard, FIPS 140-2, discussed below).

This was merely one sample, however. Two important facts about random number testing that are inherent in the statistical nature of the tests are:

- If a particular sample (or set of samples) passes all statistical tests, it may still come from a non-random generator.
- If a particular sample (or set of samples) fails some statistical test, it may still come from a random generator, although for the likelihood of a false-negative may be vanishingly small).

Thus, to properly evaluate a generator, many different samples must be analyzed. By measuring more samples and by running more tests we can increase our confidence in the test results. In the example above, we should probably run at least 1/0.0001 = 10,000 tests, each of 20,000 bits, and expect to see around one failure from a truly random generator. There are several statistical approaches for analyzing the results from many samples in order to give a probability of generator randomness. For example, first consider the passing fraction (those with P-value $\geq \alpha$,) for some test from m samples. One *confidence interval* that we use is:

$$q \pm 3\sqrt{\frac{q(1-q)}{m}} \text{ (where } q = 1-\alpha \text{)}$$

If the passing fraction falls outside this confidence interval, there is evidence that the generator *may* not be truly random.

As an example, let us assume that we test 10,000 samples from the above example and one test failed (had a P-value < α). Our passing fraction is 0.9999. In our example, the confidence level (for α = 0.0001) is calculated as 0.9996–1.0002. Since the passing fraction is within the confidence interval, the samples should be considered random (relative to this one test). If we had only had 5,000 samples with three failures, our passing fraction would be 0.9994 and the confidence interval would be 0.999476–1.00032. In this case, we would be outside the limits and probably should conclude that the samples (or the generator) may not be truly random.

This particular approach to analyzing the failure rate of multiple samples is only one example of many that can be used. Another important approach is to look at the P-values themselves. Given a large number of samples, the theory is that the P-values should have a uniform distribution between zero and one. The actual distribution can be analyzed using a chi-square function to indicate how close the distribution comes to a theoretical distribution.

7.1.2 SOME IMPORTANT TESTS

There are *many* aspects of randomness that can be tested. Accordingly, there are many tests defined and in use somewhere. A few of these are robustly defined and have achieved acceptance by the technical community. Some of the most important tests are:

- **NIST Test Suite.** The Computer Security Resource Center (CSRC) is a part of the US government's National Bureau of Standards (NIST) organization. The CSRC has defined 16 statistical tests for verifying randomness.[23] Each test addresses a different potential type of flaw in the generator. In addition to the definitions, NIST provides a suite of programs that can be used for implementing these tests.
- **FIPS-140.** This is a government cryptography standard, which includes a four-test subset of the full NIST suite with a specific numeric range of acceptability for each test.
- **The DIEHARD Tests.** These are a suite of 18 statistical tests defined by George Marsaglia of the University of Florida.[24] These are heavily used in academia and referenced in many papers on RNGs, but they are not as well documented or supported as the NIST tests. There is a great deal of overlap between the NIST tests and the DIEHARD tests, but each contains some unique tests.
- **Knuth's Tests.**[25] The tests described in Knuth's classic reference are primarily interesting from a historical or theoretical perspective since the tests described are basically covered by modern suites like the NIST. The reference is a good source for understanding the theory behind random number testing.

[23] Ibid.

[24] http://faust.irb.hr/~stipy/random/manual/html/Diehard.html

[25] Donald E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms,* Vol2, Chapter 5, Addison Wesley, 1998

In addition, there are many other tests used. Almost everyone who dabbles in random number generation has their own favorites, and, in fact, usually makes up their own tests. We fit this profile, but are limiting ourselves in this document to discussing and presenting results from the two most reputable test suites: NIST (including FIPS 140) and DIEHARD.

When testing an RNG, it may also be appropriate to construct additional tests based on the design of a particular RNG. For example, if an RNG was constructed by flipping ten coins in order then repeating such that every tenth bit was produced using the same coin, it would be appropriate to apply tests that look at every tenth bit.

7.1.3 NIST TEST SUITE

The NIST test suite is the best documented and most formally analyzed of the publicly available tests. Following is a very terse summary of what each NIST test does. Each of these tests calculates a P-value, which identifies the probability that a truly random generator could produce a string less random than the sample.

The NIST tests are

- Monobit frequency test. Analyze the number of ones and zeros.
- Frequency test within a block. Analyze the frequency of ones within M-bit blocks.
- Runs test. Analyze the frequencies of runs (consecutive ones or zeros) of various lengths.
- Longest Runs test. Analyze the probability of occurrence of the longest run of ones.
- Matrix Rank test. Analyze the chi-squared statistics of the rank of disjoint sub-matrices chosen from the sample.
- Discrete Fourier (Spectral) test. Perform a DFT on the data (convert the data to the frequency domain) and analyze for periodic frequencies.
- Non-overlapping Template Matching test. Analyze the number of times specified bit patterns (templates) occur in the bitstream in an non-overlapped fashion.
- Overlapping Template Matching test. Same as the non-overlapping test except that the patterns can overlap.
- Maurer's "Universal Statistical" test. Analyze the number of bits between matching patterns. This test identifies whether the sequence can be significantly compressed without loss of information.
- Lempel-Ziv compression test. The sample is compressed using a standard compression algorithm. The degree of compression indicates the lack of randomness of the sample.
- Linear complexity test. Calculate and analyze the LFSR that could generate the bitstream.
- Serial test. Analyze the frequency of overlapping *m*-bit patterns across the sample.
- Approximate Entropy test. Analyze the entropy of blocks chosen from the bitstream.
- Cumulative Sum test. Analyze the maximum cumulative sum computed in a random walk.
- Random Excursions test. Analyze the frequency of visits to states in a cumulative sum random walk.

Beyond the test definition, NIST provides a source-code package for these tests. While the user can define the test parameters, NIST provides recommended sample sizes along with a target α value of 0.01. This α defines a fairly strenuous level of randomness. Pass/fail statistics are produced for each sample (based on the calculated P-value compared to α) along with an analysis of the multiple P-values for each test. We use these tests and the NIST recommended values in our testing procedure.

7.1.4 FIPS 140

The US government's FIPS (Federal Information Processing Standard) 140 defines formal security requirements for *cryptographic modules*. This standard is supported by a government-provided Cryptographic Module Validation Program (CMVP), which validates "devices" as conforming to FIPS 140 (or other relevant cryptography standards). The latest version is FIPS 140-2[26], which succeeded FIPS 140-1[27]. The scope and requirements of FIPS 140-2 go far beyond random number generators, but RNGs are covered since they are an essential component of cryptographic modules.

Both FIPS versions define the same four RNG tests with the same test sample size of 20,000 bits. Two of the tests are from the full NIST test suite. The other two are variations on two tests from the NIST suite. These four tests are among the simpler tests in the NIST suite (since the FIPS tests are designed to be run at device startup). FIPS 140 defines "pass/fail" statistical limits for each test. The only difference between 140-2 and 140-1 (relative to random number testing) is that FIPS 140-2 has tougher "pass/fail" limits for the statistical tests.

The 140-1 pass/fail limits are calculated for an α of 0.000001, and the 140-2 limits are supposed to be calculated for an α of 0.0001 (remember, a higher α is a tougher criteria). This means, for example, that, a truly random sample should (statistically) fail the FIPS 140-2 tests 1 out of 10,000 samples. Note that the full NIST suite uses a much higher α value of 0.01. While the NIST test is helpful for testing RNGs, it is not suitable for production-line testing since a perfect RNG would fail the tests one percent of the time.

The four FIPS 140 tests are:

- **Bit frequency test.** Counts the number of 1 bits in the sample. This number must be within specified limits.
- **"Poker" test.** Calculates a statistic based on the frequency distribution of independent 4-bit chunks of the sample. This statistical value must be within specified limits.
- **Runs test.** Counts the number of runs of various lengths that occur in the sample. A "run" is a string of consecutive ones. The number of runs for lengths up to six or more must be within specified limits.
- **Long runs test.** Defines a maximum run size, *n* (34 for 140-1), of consecutive ones or zeroes. To pass, the maximum run length in the sample must be less than *n*.

Problems with the FIPS 140-2 Definition

Our analysis shows that the limits used for the "runs" and "long runs" tests in FIPS 140-*2* are wrong; that is, they aren't consistent with the specified α of 0.0001. A NIST representative has

[26] http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf.

[27] http://csrc.nist.gov/publications/fips/fips1401.pdf

confirmed this fact.[28] They have not yet identified, however, a plan to get them publicly corrected. Accordingly, our summary data reports do not show any results from these particular FIPS-2 tests. The runs and long runs tests are reported for the FIPS 140-1 limits, however.

7.1.5 DIEHARD TESTS

TBA

7.2 OUR TESTING METHODOLOGY

Extensive testing of the VIA C3 Nehemiah RNG feature has been performed by Centaur Technology. Tests were performed on many different chips running on several different PC platforms running Red Hat Linux 7.1 (kernel version 2.4.2-2). The general testing approach was:

- Each test (or set of tests) was run on a "large sample". This sample typically consisted of from millions to tens of billions of contiguous bits collected from a single part (and system).
- The same test was run on *many* large samples. The high speed of the VIA C3 Nehemiah RNG allows this massive sampling. This is in contrast to reported RNG testing for other random number sources; these other sources have more limited sample sizes because of their limited RNG speeds.[29]
- Each specific large sample was used by only the particular tests that were enabled while the data were being collected. Thus, different tests often used different sets of generated data. Not using the same exact data for different tests is acceptable because of the huge sample size, and it may be statistically preferable.
- Most tests used the default settings for DC bias. A few tests were run and analyzed at alternate settings.
- Most tests were run at ambient conditions. A few were run and analyzed at worst case temperature conditions.
- All tests were run across all settings of the EDX hardware rate divisor. The higher divisors were tested more than the lower divisors, however.
- The primary tests run were the FIPS 140-1, the FIPS140-2, the NIST, and the DIEHARD tests. Some additional tests were run with limited samples. . For example, we studied the use of a SHA hash on the generated bits.

As an interesting comparison. the same tests were also run on "random numbers" from other sources. One source is the standard Linux software random number generator—/dev/urandom—running on the same PC platforms. This generator gathers environmental noise from device drivers and other sources into an entropy pool. From this entropy pool random numbers are created using a SHA-1 hash.

[28] Private communications with NIST personnel from November, 2002.

[29] For example, Jun and Kocher reported tests "on at least 80 megabits of continuous RNG output" for the Intel 82802 RNG.

A second alternate source is random data provided by www.random.org. This data is derived on atmospheric noise as sampled as electrical noise (static) on a radio receiver.

7.3 SOME SUMMARY RESULTS

Following is a summary of some of the more important test results.

7.3.1 FIPS-140 TESTS

(this data will be provided by special request or in a subsequent version)

7.3.2 DIEHARD TESTS

(this data will be provided by special request or in a subsequent version)

7.3.3 NIST TESTS

(this data will be provided by special request or in a subsequent version)

7.4 TESTING CONCLUSIONS

(TBA)